Algorithmique

Alexandre Duret-Lutz
adl@lrde.epita.fr

2 août 2010

Troisième partie III

Structures de données

Types abstraits, structures de données

Types abstraits et structures de données

Un type abstrait est une spécification mathématique d'un ensemble de données et de l'ensemble des opérations qu'on peut y effectuer. C'est un contrat qu'une structure doit implémenter.

Par exemple le type abstrait « Pile » peut être réalisé (implémenté) avec une structure de liste simplement chaînée ou avec un tableau.

Pour décrire un algorithme, on peut n'utiliser que des types abstraits. L'utilisation d'un objet pile spécifie clairement les comportements attendus.

Avant d'implémenter l'algorithme il faudra faire le choix d'une structure de données pour le type abstrait.

La complexité des opérations sur un type abstrait peut dépendre de son implémentation.

Représentations d'ensembles

Des représentations d'ensembles de valeurs

- Séquences
  - Tableau, vecteur (*array*, *vector*)
  - Liste (*list*)
  - Pile (*stack*)
  - File (*queue*)
    - File de priorité (*priority queue*).
  - File à double entrée (*deque*, pronounce like *deck*)
- Tableaux associatifs, structures de recherche
  - Table de hachage (*hash table*)
  - Arbre binaire de recherche équilibré (*self-balancing binary serach tree*)
  - Liste à enjambement (*skip list*)
  - Diagramme de décision binaire (*binary decision diagram*)

On choisit une structure de donnée en fonctions des opérations qu'on a besoin de faire sur l'ensemble représenté et de la complexité des opérations correspondantes sur la structure de donnée.

Des opérations sur les ensembles

$v \leftarrow$Access(S,k) Retourne la valeur du k^{e} élément.

$p \leftarrow$Search(S,v) Retourne un pointeur (ou indice) sur un élément de S dont la valeur est v.

Insert(S,x) Ajoute l'élément x à S.

Delete(S,p) Efface l'élément de S à la position (pointeur ou indice) p.

$v \leftarrow$Minimum(S) Retourne le minimum de S.

$v \leftarrow$Maximum(S) Retourne le maximum de S.

$p' \leftarrow$Successor(S,p) Retourne le successeur (la plus petite valeur supérieure) dans S de l'élément indiqué par p.

$p' \leftarrow$Predecessor(S,p) Devinez.

Bien sûr il nous arrive aussi de vouloir trier une séquence, concaténer (union) deux ensembles, ou à l'inverse en couper un en deux, etc.

Tableaux

On ne les présente plus...

opération	tableau non trié	tableau trié
$v \leftarrow$Access(S,k)	$\Theta(1)$	$\Theta(1)$
$p \leftarrow$Search(S,v)	$\mathrm{O}(n)$	$\mathrm{O}(\log n)$
Insert(S,x)	$\Theta(1)$	$\mathrm{O}(n)$
Delete(S,p)	$\Theta(1)$ [1]	$\mathrm{O}(n)$
$v \leftarrow$Minimum(S)	$\Theta(n)$	$\Theta(1)$
$v \leftarrow$Maximum(S)	$\Theta(n)$	$\Theta(1)$
$p' \leftarrow$Successor(S,p)	$\Theta(n)$	$\Theta(1)$
$p' \leftarrow$Predecessor(S,p)	$\Theta(n)$	$\Theta(1)$

1. L'ordre n'important pas on remplace l'élément supprimé par le dernier du tableau.

Tableaux dynamiques (1/2)

Leur taille peut varier.

Besoin de faire `realloc()` en C quand il n'y a plus de cases libres. En C++ `std::vector` fait le `realloc()` lui-même. La réallocation d'un tableau demande $\Theta(n)$ opérations car il faut le copier. L'insertion, normallement en $\Theta(1)$, devient en $\Theta(n)$ quand cette réallocation est nécessaire.

On veut étudier la **complexité amortie** d'une insertion dans une séquence d'insertions.

Lors d'une réallocation, on s'arrange pour allouer plusieurs nouvelles cases d'un coup afin que le surcoût de cette réallocation reste marginal par rapport au nombre des insertions.
Plusieurs façons de choisir la nouvelle taille d'un tableau sont possibles et donnent des complexités différentes.

Tableaux dynamiques (2/2)

On considère une insertion qui provoque une réallocation dans un tableau de taille n, avec deux façons d'agrandir le tableau :

Agrandissement constant de k cases. Il y a donc une réallocation toutes les k cases : le côut moyen des k dernières insertions est

$$\frac{(k-1)\Theta(1)+1\Theta(n)}{k} = \Theta(n)$$

Doublement de la taille du tableau. Depuis la dernière réallocation a il y a eu $n/2-1$ insertions en $\Theta(1)$ puis une insertion en $\Theta(n)$. Le côut moyen des $n/2$ dernières insertions est

$$\frac{(n/2-1)\Theta(1)+1\Theta(n)}{n/2} = \frac{\Theta(n)+\Theta(n)}{n} = \Theta(1)$$

On parle de complexité en « $\Theta(1)$ **amorti** » quand l'opération est en $\Theta(1)$ la plupart du temps, et que le surcoût des cas lents peut être étalé (amorti) sur les cas où l'opération est rapide.

Listes

On distingue surtout les liste simplement chaînées (où l'on ne connaît que l'élément suivant) des listes doublement chaînées (où l'on connaît aussi l'élément précédent).

	liste simpl.ch.		liste doubl.ch.	
opération	non triée	triée	non triée	triée
$v \leftarrow$Access(S,k)	$\mathrm{O}(n)$	$\mathrm{O}(n)$	$\mathrm{O}(n)$	$\mathrm{O}(n)$
$p \leftarrow$Search(S,v)	$\mathrm{O}(n)$	$\mathrm{O}(n)$	$\mathrm{O}(n)$	$\mathrm{O}(n)$
Insert(S,x)	$\Theta(1)$	$\mathrm{O}(n)$	$\Theta(1)$	$\mathrm{O}(n)$
Delete(S,p)	$\mathrm{O}(n)$ [1]	$\mathrm{O}(n)$ [1]	$\Theta(1)$	$\Theta(1)$
$v \leftarrow$Minimum(S)	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
$v \leftarrow$Maximum(S)	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
$p' \leftarrow$Successor(S,p)	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
$p' \leftarrow$Predecessor(S,p)	$\Theta(n)$	$\mathrm{O}(n)$	$\Theta(n)$	$\Theta(1)$

1. En vrai on s'arrange pour connaître l'élément précédent et rester en $\Theta(1)$.

Piles et files

Pile (*stack*) Séquence dans laquelle les insertions et suppressions sont toujours faites à une extrémité (toujours la même). Insert() et Delete() sont généralement appelés Push() et Pop(). LIFO

- Généralement implémentée au dessus d'un tableau. Ajout et suppression en queue en $\Theta(1)$.

File (*queue*) Séquence dans laquelle les insertions sont toujours faites à la même extrémité, et les suppressions à l'autre. Insert() et Delete() sont généralement appelés Push() et Pop(). FIFO

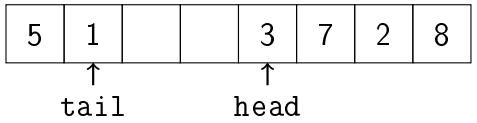
- Généralement implémentée au dessus d'une liste simplement chaînée. Ajout en queue en $\Theta(1)$, suppression en tête en $\Theta(1)$.

File à double entrée (*deque*) Les insertions et suppressions peuvent être effectuées à l'une ou l'autre des extrémités.

- Peut être implémentée au dessus d'une liste doublement chaînée. Ajouts et suppressions en $\Theta(1)$.

Files bornées et tableaux circulaires

Si la taille d'une file (à simple ou double entrée) est bornée, elle peut être implémentée efficacement par un tampon circulaire.



Dans ce cas l'accès à la k^{e} se fait en $\Theta(1)$ au lieu de $\Theta(n)$ sur une liste.

$$\text{Access}(S, k) = A[(head + k - 1) \mod n]$$

En contrepartie, Insert() et Erase() au rang r deviennent en $\mathrm{O}(\min(r, n - r))$ au lieu $\mathrm{O}(1)$.

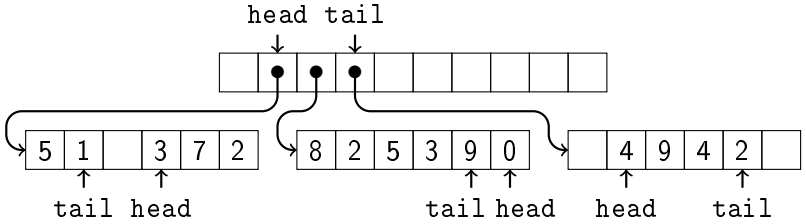
Comment étendre ce schéma à des files non bornées ?

Files non-bornées et tableaux circulaires

Ajouter un élément dans un tableau circulaire plein.

1re approche Augmenter la taille du tableau. En pratique : nouvelle allocation plus copie. L'insertion devient en $\Theta(n)$ quand cela arrive. Mais la complexité peut rester en « $\Theta(1)$ amorti ».

2e approche Gérer un tableau circulaire de tableaux circulaires de taille constante. Seules les extrémités ne sont pas pleines.



Encore du « $\Theta(1)$ amorti » car il faut parfois réallouer l'index, mais c'est plus rare. C'est l'implémentation de `std::deque` en C++.

Files de priorité

L'élément retiré d'une file est toujours le plus grand. On parle parfois de « *Largest In, First Out* » (ne pas confondre avec LIFO = *Last In First Out*).

- Si la file de priorité est réalisée à l'aide d'une liste triée, alors on a Push() en $\mathrm{O}(n)$ et Pop() $\Theta(1)$.
- Si la file de priorité est réalisée à l'aide d'un tas, alors on a Push() en $\mathrm{O}(\log n)$ et Pop() $\mathrm{O}(\log n)$.

  On sait faire un Pop() sur un tas : comme dans le tri par tas on retire la première valeur du tas, on la remplace par la dernière, et on appelle Heapify() pour rétablir la structure de tas.

  Comment faire un Push() ?

Files de priorité : Insertion dans un tas

Entrée : Un tableau $A[1..m]$ respectant la propriété de tas, une valeur v à y insérer,
Sortie : Un tableau $A[1..m+1]$ respectant la propriété de tas et contenant v.

HeapInsert(A, m, v)
1 $i \leftarrow m + 1$
2 $A[i] \leftarrow v$
3 while $i > 1$ and $A[Parent(i)] < A[i]$ do
4 $\quad A[Parent(i)] \leftrightarrow A[i]$
5 $\quad i \leftarrow Parent(i)$

Au pire on fait un nombre d'opérations proportionnel à la hauteur du tas, donc $T(n) = \mathrm{O}(\log n)$.

Tableaux associatifs, structures de recherche

Tableau associatif

Un tableau associatif (ou encore dictionnaire ou table d'association) est un type abstrait pouvant être vu comme un généralisation de la notion de tableau à des indices non consécutifs et pas forcément entiers. Les indices sont ici appelés des clefs. Les clefs peuvent être associées à des valeurs « satellites », exactement comme dans les tris.

Opérations typiques :

- Ajout
- Suppression
- Recherche (par clef).
- Modification (des données satellites).

La présentation des structures ne montre pas les données satellites. Il faut supposer qu'elles accompagnent la clef mais n'interviennent pas dans les opérations sur les clefs (e.g. comparaisons).

Tables de hachage

Le but : représenter un ensemble $\mathcal{F}$ d'éléments quelconques (les clefs), disons un sous-ensemble d'un domaine $\mathcal{K}$. On veut tester rapidement l'appartenance à cet ensemble.

Si $\mathcal{K} = \mathbb{N}$, alors on peut utiliser un tableau pour représenter $\mathcal{F}$. Par exemple $n \in \mathcal{F}$ ssi $A[n] \neq 0$ (le premier indice du tableau A est 0).
Cependant si $\max(\mathcal{F})$ est grand le tableau va occuper beaucoup de place même si $|\mathcal{F}|$ est petit.
On s'inquiète aussi du cas où $\mathcal{K}$ est autre chose que $\mathbb{N}$.

L'idée : pour $\mathcal{F} \subseteq \mathcal{K}$ quelconque, on se donne une fonction $f : \mathcal{K} \mapsto [\![0, m]\!]$. On teste alors l'appartenance avec $x \in \mathcal{F}$ ssi $A[f(x)] \neq 0$.
Ce test est correct si la fonction f est injective (de telles fonctions n'existent que si $m - 1 \geq |\mathcal{K}|$).

Ces tests l'appartenance sont en $\Theta(1)$.

Injectivité dans $\mathcal{K}$ ou dans $\mathcal{F}$

Soit $\mathcal{F} = \{$`"chat"`, `"chien"`, `"oie"`, `"poule"`$\}$ à ramener dans $[\![0, 30]\!]$.
Prenons comme fonction $f(mot) = (mot[2] - $`'a'`$)$.

Cette fonction distingue les mots de $\mathcal{F}$ par leur troisième lettre.
$f($`chat`$) = 0$
$f($`chien`$) = 8$
$f($`oie`$) = 4$
$f($`poule`$) = 20$
Mais elle n'est pas injective dans $\mathcal{K}$:
$f($`loup`$) = 20$
On s'en sort en représentant l'élément dans le tableau : $x \in \mathcal{F}$ ssi $A[f(x)] = x$.
Pour l'injectivité dans $\mathcal{F}$ on veut donc seulement $m \geq |\mathcal{F}|$.

i	$A[i]$
0	`chat`
1	/
2	/
3	/
4	`oie`
⋮	/
8	`chien`
⋮	/
20	`poule`
⋮	/

Rareté des fonctions injectives

Les fonction f injectives ne sont pas faciles à trouver « par hasard ».

Soit un ensemble $\mathcal{F}$ de $n = 30$ éléments qu'on cherche à représenter dans un tableau de $m = 40$ entrées.

Il existe $40^{30} \approx 10^{48}$ fonctions de $\mathcal{F}$ dans $[\![0, m-1]\!]$. Parmi ces fonctions, seules $40 \cdot 39 \cdots 11 = 40!/10! \approx 2.10^{41}$ sont injectives. On a donc une chance sur 5 millions de trouver une fonction injective pour cet exemple.

Exemple typique de la rareté des fonctions injective : le paradoxe des anniversaires. Il suffit de réunir 23 personne pour avoir plus d'une chance sur deux que deux de ces personnes soient nées le même jour. Pourtant la fonction *anniversaire* a 365 choix possibles !

`http://fr.wikipedia.org/wiki/Paradoxe_des_anniversaires`

GPerf

Lorsque l'ensemble $\mathcal{F}$ est connu à l'avance, il est possible de trouver (algorithmiquement) une fonction f qui transforme $\mathcal{F}$ en $[\![0, m]\!]$ avec $m - 1 \geq \mathcal{F}$ sans doublons. On parle de fonction de hachage parfaite. Une telle fonction est minimale si $m - 1 = |\mathcal{F}|$.

L'outil GNU `gperf` est dédié à cette tâche pour des ensembles de chaînes. Il prend en entrée une liste de mots à reconnaître, une valeur m, et fournit en sortie un fichier C contenant la fonction f et le tableau A avec les éléments dans l'ensemble rangés aux places indiquées par f.

D'autres outils existent, p.ex. CMPH (C Minimal Perfect Hashing Library).

Hachage avec chaînage

Lorsqu'une fonction de hachage parfaite n'est pas disponible (par exemple m est trop petit, ou bien l'ensemble $\mathcal{F}$ est sans cesse modifié) deux éléments peuvent devoir être rangés au même indice et l'on doit gérer ces collision.

Comme dans le tris par paquet, la façon la plus simple et de conserver une liste de valeurs possibles pour chaque indice du tableau.

Reprenons $\mathcal{F} = \{$`"chat"`, `"chien"`, `"oie"`, `"poule"`, `"loup"`$\}$.
Alors $x \in \mathcal{F}$ ssi $Search(A[f(x)], x) \neq 0$.
La recherche ne se fait plus en $\Theta(1)$ parce qu'il faut parcourir une liste. Dans le pire des cas, la liste fait $n = |\mathcal{F}|$ éléments, dans le meilleur on espère n/m éléments.

i	$A[i]$
0	`chat`
1	/
2	/
3	/
4	`oie`
⋮	/
8	`chien`
⋮	/
20	`poule, loup`
⋮	/

Hachage avec chaînage : hachage uniforme

On parle de hachage uniforme simple lorsqu'on suppose que la fonction de hachage f répartit les clefs uniformément dans $[\![0, m-1]\!]$.

Sous l'hypothèse de hachage uniforme, la recherche se fait en $\Theta(1 + n/m)$.

Si l'on s'arrange pour que la taille m de la table de hachage soit proportionelle à n. Alors $n = \mathrm{O}(m)$, $n/m = \mathrm{O}(1)$, et la recherche se fait en $\Theta(1)$.

L'insertion se fait alors en $\Theta(1)$ amorti (car réallocation) et la suppression en $\Theta(1)$. La réallocation demande de changer de fonction de hachage (puisque m change) et de déplacer tous les éléments de $\mathcal{F}$ à leur nouvelle place dans le tableau.

Exemples de fonctions de hachage : La division

$f(x) = x \mod m$
On évitera une puissance de deux telle que $m = 256$ car cela revient à ignorer les 8 bits les plus faibles de x, souvent ceux qui varient le plus. On conseille de prendre pour m un nombre premier assez éloigné d'une puissance de 2.
Par exemple pour représenter 3000 éléments avec une moyenne de deux éléments par liste, on peut choisir $m = 1543$.
Une implémentation de table de hachage par cette méthode contient typiquement une liste de nombres premiers à utiliser au fur et à mesure que la table s'agrandit. Par exemple si on double la taille de la table chaque fois qu'on la réalloue : 53, 97, 193, 389, 769, 1543, 3079, 6151, 12289, 24593, 49157, 98317, 196613, 393241, etc.

La classe `std::hash_map` du C++ fait cela avec la fonction $f(x) = g(x) \mod m$ où g est fournie par l'utilisateur (pour convertir n'importe quel type en entier) et m suit la liste ci-dessus.

Ex. de fonctions de hachage : La multiplication

$f(x) = \lfloor m(xa \mod 1) \rfloor$ où $xa \mod 1 = xa - \lfloor xa \rfloor$ et $0 < a < 1$.

On vise un choix de a qui distribue bien les valeurs de $xa \mod 1$ dans $[\![0, 1[\![$.
Ici le choix de m n'a pas d'importance, autrement dit on peut faire croître la taille du tableau sans contrainte.
Choisir $m = 2^p$ permet de faire des calculs entiers. Si la machine possède des entiers de w bits, on calcule $\lfloor a \cdot 2^w \rfloor \times k$, ce qui nous donne un résultat de $2w$ bits dont la deuxième moitié représente la partie fractionnaire. On garde les p premiers bits de cette moitié.

Peut être vu comme une généralisation de la division : $a = 1/b$, mais en pratique plus rapide (multiplication moins coûteuse).

Choix de a ? Knuth suggère une valeur de a pas trop éloignée d'un nombre irrationnel (par exemple le nombre d'or $(\sqrt{5} - 1)/2$) et tel que $a \cdot 2^w$ et 2^w soit premiers entre eux.

Effet d'avalanche pour les fonctions d'hachage

L'effet d'avalanche est une propriété désirable en cryptographie, mais elle est aussi utile pour une table de hachage.
Si $y = f(x)$ est la valeur de hachage pour x, on souhaite qu'une petite modification de x induise une grande modification de y.

Le critère d'avalanche stricte (*Strict Avalanche Criterion*) : « pour toute inversion d'un seul bit en entrée alors chaque bit en sortie a une chance sur deux d'être modifié ».

MD5 est une fonction de hachage (coûteuse) qui respecte ce critère.

```
~ % echo "Bonjour tout le monde" | md5sum -
52e1281d5b6885323e40a9789e1a8c8d  -
~ % echo "bonjour tout le monde" | md5sum -
a6fabe06142f2bb0ed6b996a9cd50221  -
```

Attention, aujourd'hui on sait générer rapidement (quelques secs sur une machine de bureau) une paire de messages qui ont la même clef MD5.

Autre exemple de fonction de hachage pour entier

```
/* sizeof int == 32 */
int wang32_hash(int key)
{
  key += ~(key << 15);
  key ^=  (key >> 10);
  key +=  (key << 3);
  key ^=  (key >> 6);
  key += ~(key << 11);
  key ^=  (key >> 16);
  return key;
}
```

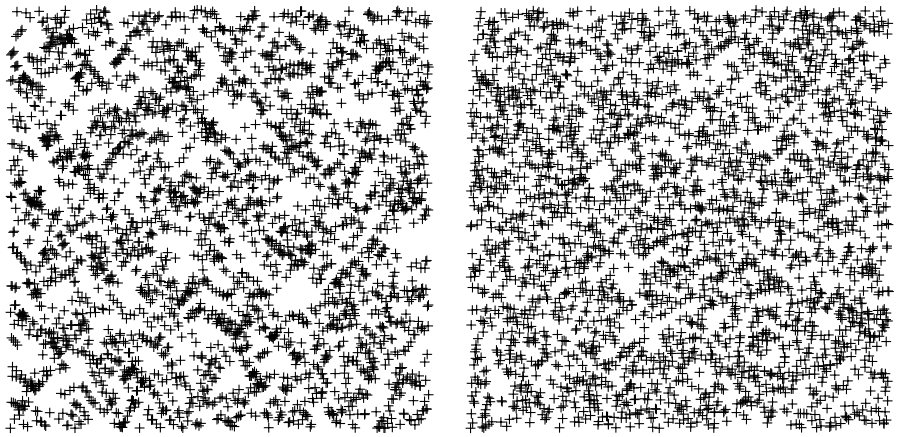
On peut combiner cette fonction plusieurs fois pour hacher quelque chose de plus gros. Par exemple

`wh(wh(wh(s[0])^s[1])^s[2])`

L'« avalanche » prend plus de sens lorsque la fonction est appliquée ainsi plusieurs fois.

Comment vérifier une telle fonction ? Exercice : tracez les 30000 premières valeurs de $y = wh(x)$ avec Gnuplot. On voit des motifs de « zones vierges » apparaître. Tracez $y = wh(wh(x))$: ça paraît beaucoup plus aléatoire.

$y = wh(x)$ et $y = wh(wh(x))$



Adressage ouvert

Tous les éléments sont stockés dans la table, sans liste ni pointeurs. Résolution des collisions sans chaînage : on doit juste examiner plusieurs positions pour trouver la bonne.

Une fonction de hachage pour une clef x prend aussi une itération i.

Pour insérer une valeur, on teste si $A[f(x,0)]$ est libre, sinon $A[f(x,1)]$, sinon $A[f(x,2)]$, etc. On dit qu'on sonde différentes positions.
La fonction f doit être telle que $f(x,i)$ parcourt $[\![0,m[\![$ lorsque i parcourt $[\![0,m[\![$. L'ordre de parcours dépend de la clef x.

La recherche se fait de la même façon jusqu'à ce qu'on tombe sur une case vide.

Danger à la suppression : pourquoi ne peut-on pas remplacer vider la case bêtement ?

Fonctions de hachage pour adressage ouvert

Sondage linéaire $h(x,i) = (h'(x) + i) \mod m$
Problème : plus une séquence d'emplacements occupés est longue, plus elle risque d'être allongée, augmentant les temps de recherche. (On peut chercher à remplacer $+i$ par $+ci$ si c et m sont premiers entre eux, mais cela change rien : des séquences de cases espacés de c occupées seront de plus en plus longues.)

Sondage quadratique $h(x,i) = (h'(x) + c_1 i + c_2 i^2) \mod m$
C'est mieux, mais ici aussi, le premier sondage détermine la séquence toute entière. $h(x,0) = h(x',0) \implies h(x,i) = h(x',i)$.

Double hachage $h(x,i) = (h_1(x) + i h_2(x)) \mod m$
Cette fois-ci $h(x,0) = h(x',0) \not\Rightarrow (x,i) = h(x',i)$.

Complexité des recherches dans l'adressage ouvert

Le nombre de sondages attendus dans une recherche infructueuse est $1/(1-n/m)$ si l'on suppose le hachage uniforme (i.e., toutes les séquences de parcours de $[\![1,m[\![$ peuvent apparaître avec la même probabilité).

De même insérer demande $1/(1-n/m)$ sondages en moyenne.

Si n/m est constant, on en déduit que l'insertion, la recherche et la suppression sont en $\Theta(1)$. En pratique m n'est bien sûr pas changé aussi souvent que n.

L'intérêt de l'adressage ouvert sur le chaînage est de supprimer les pointeurs. Cela donne plus de mémoire et permet de stocker des tables plus grandes (plus grand m). La contrepartie : c'est plus lent en pratique.

Taille critique des tableaux de hachage

L'étude du paradoxe des anniversaires nous apprend que si la table de hachage peut représenter N entrées le nombre d'éléments nécessaires pour avoir une probabilité p de collisions est d'environ

$$n(p, N) \approx \sqrt{2N \ln\left(\frac{1}{1-p}\right)}$$

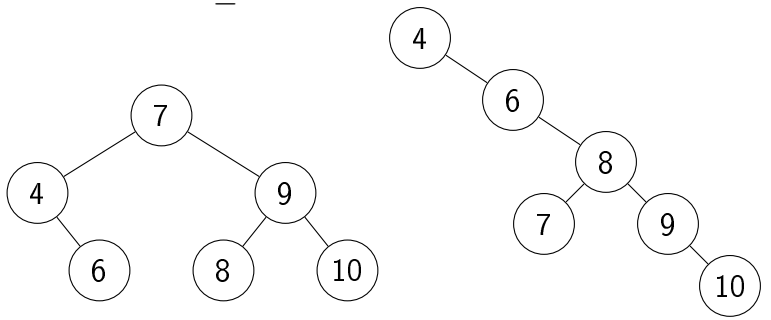
On retiendra surtout :

$$n(0.5, N) \approx 1.177\sqrt{N} = \Theta(\sqrt{N})$$

C'est-à-dire qu'a partir de $\sqrt{N}$ valeurs dans une table de hachage (hachée uniformément) on a à peu près une chance sur deux d'avoir une collision.

Arbres binaires de recherche

Un arbre binaire dont les nœuds sont étiquetés par des valeur est dit de recherche si pour un nœud r étiqueté par v toutes les étiquettes du sous-arbre gauche de r sont $\leq v$, et toutes celles du du sous-arbre droit de r sont $\geq v$.



Tous les ABR ne sont pas équilibrés. La complexité de la recherche est $\mathrm{O}(h)$ avec h la hauteur de l'arbre. Recherche du minimum et maximum en $\mathrm{O}(h)$ aussi.

Ordre infixe

Parcourir l'arbre dans l'ordre infixe permet de visiter les clefs dans l'ordre.

```
InfixPrint(T, z)
1  if LeftChild(z) ≠ NIL then
2      InfixPrint(T, LeftChild(z))
3  print key(z)
4  if RightChild(z) ≠ NIL then
5      InfixPrint(T, RightChild(z))
```

```
PrefixPrint(T, z)
1  print key(z)
2  if LeftChild(z) ≠ NIL then
3      InfixPrint(T, LeftChild(z))
4  if RightChild(z) ≠ NIL then
5      InfixPrint(T, RightChild(z))
```

```
SuffixPrint(T, z)
1  if LeftChild(z) ≠ NIL then
2      InfixPrint(T, LeftChild(z))
3  if RightChild(z) ≠ NIL then
4      InfixPrint(T, RightChild(z))
5  print key(z)
```


Insérer dans un ABR est facile

Entrée : un ABR T et un enregistrement z à y ajouter
Sortie : l'ABR T contenant z

```
TreeInsert(T, z)
 1  y ← NIL
 2  x ← Root(T)
 3  while x ≠ NIL do
 4      y ← x
 5      if key(z) < key(x)
 6          then x ← LeftChild(x)
 7          else x ← RightChild(x)
 8  Parent(z) ← y
 9  if y = NIL then
10      Root(T) ← z
11  else
12      if key(z) < key(y)
13          then LeftChild(y) ← z
14          else RightChild(y) ← z
```

$T(h) = \mathrm{O}(h)$

Suppression (1/2)

Trois cas à considérer :

- La suppression d'une feuille de l'arbre : facile.
- La suppression d'un nœud qui n'a qu'un fils : facile aussi.
- La suppression d'un nœud avec deux fils : on le remplace par son successeur, i.e. le minimum de l'arbre droit, qui aura été à son tour supprimé de sa place plus bas dans l'arbre.

Suppression (2/2)

```
TreeDelete(T, z)
 1  x ← NIL
 2  if LeftChild(z) = NIL or RightChild(z) = NIL
 3     then y ← z
 4     else y ← TreeSuccessor(z)
 5  if LeftChild(y) ≠ NIL
 6     then x ← LeftChild(y)
 7     else x ← RightChild(y)
 8  if x ≠ NIL then Parent(x) ← Parent(y)
 9  if Parent(y) = NIL then
10     Root(T) ← x
11  else
12     if y = LeftChild(Parent(y))
13        then LeftChild(Parent(y)) ← x
14        else RightChild(Parent(y)) ← x
15  if y ≠ z then key(z) ← key(y)
```


Le problème des ABR

Insertion, Suppression, Recherche, Prédécesseur, Successeur, Minimum et Maximum s'effectuent tous en $\mathrm{O}(h)$ et

$$\underbrace{\lfloor \log n \rfloor}_{\text{cas équilibré}} \leq h \leq \underbrace{n}_{\text{cas déséquilibré}}$$

Donc ces algorithmes sont donc en $\mathrm{O}(n)$...

On peut cependant montrer que la hauteur moyenne d'un ABR construit aléatoirement (on fait la moyenne de tous les ordres d'insertion possible pour une séquence donnée) est en $\Theta(\log n)$.

On peut modifier les opérations d'insertion et de suppression pour essayer de préserver la nature équilibrée d'un arbre, et donc améliorer les performances de ces algorithmes.

Arbres rouge et noir

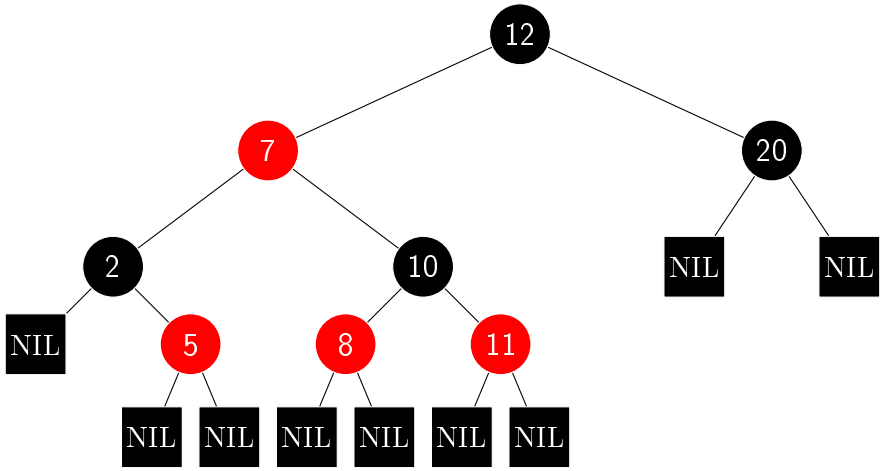
Ce sont des ABR dans lesquels chaque nœud possède un bit indiquant sa couleur : rouge ou noir. Des contraintes sur les couleurs empêchent qu'une branche soit plus de deux fois plus longue qu'une autre. L'arbre est alors approximativement équilibré.

Les contraintes :

- Chaque nœud est soit rouge, soit noir.
- La racine et les feuilles (NIL) sont noires.
- Les deux fils d'un nœud rouge sont noirs.
- Tous les chemins reliant un nœud à une feuille (de ses descendants) contient le même nombre de nœuds noirs.

La hauteur noire d'un nœud x, notée $hn(x)$ est le nombre de nœuds noirs entre x (exclu) et une feuille (inclue).

Exemple d'ARN



Propriété

Un arbre rouge et noir avec n nœuds internes possède une hauteur au plus égale à $\lfloor 2\log(n+1) \rfloor$.

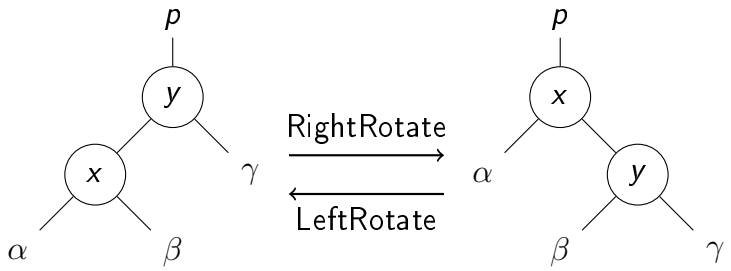
- Si on fait disparaître les nœuds rouges, les nœuds internes noirs ont entre 2 et 4 fils, et toutes les branches de l'arbre ont toutes la même longueur h'.
- La hauteur de l'arbre complet est $h \leq 2h'$ car il ne peut pas y avoir plus de nœuds rouges que de nœuds noirs sur une branche.
- Le nombre de feuilles des deux arbres est $n+1$.
  On a donc

$$n+1 \geq 2^{h'} \quad \Longrightarrow \quad \log(n+1) \geq h' \geq h/2 \quad \Longrightarrow \quad h \leq 2\log(n+1)$$

D'autre part la longueur minimale d'une branche est $\log(n+1)$ (moitié de la hauteur). En conséquence, Search, Minimum, Maximum, Successor et Predecessor sont en $\Theta(\log n)$. Ce n'est pas évident pour Insert et Delete.

Rotations

Insérer avec `TreeInsert` ne préserve pas les propriétés des ARN. On peut rétablir ces propriétés en changeant les couleurs des nœuds et en effectuant des rotations localement.



Ces rotations préservent l'ordre infixe.

Rotation gauche

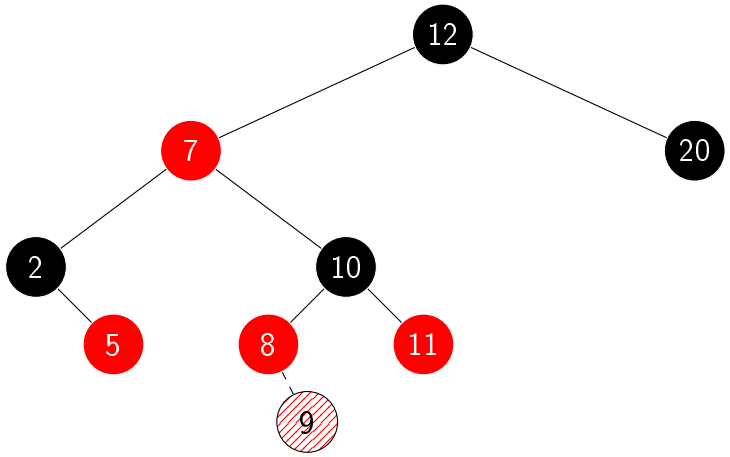
```
LeftRotate(T, x)
 1  y ← RightChild(x)
 2  β ← LeftChild(y)
 3  RightChild(x) ← β
 4  if β ≠ NIL then Parent(β) ← x
 5  p ← Parent(x)
 6  Parent(y) ← p
 7  if p = NIL
 8      then Root(T) ← y
 9      else if x = LeftChild(p)
10          then LeftChild(p) ← y
11          else RightChild(p) ← y
12  LeftChild(y) ← x
13  Parent(x) ← y
```

$T(n) = \Theta(1)$

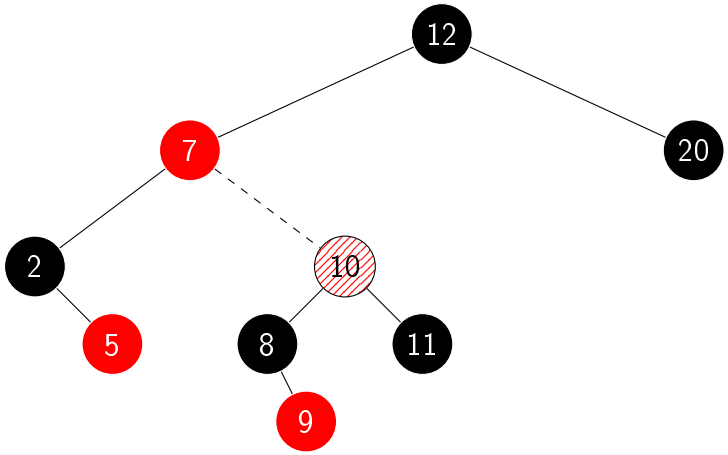
Insertion dans un ARN

- On insère le nœud dans l'arbre avec la couleur rouge. La propriété « les fils d'un nœuds rouge sont noirs » peut être violée chez le père du nœud inséré.
- On corrige cette violation en la faisant remonter par recolorations, jusqu'à pouvoir la corriger par rotation (et peut-être recoloration).

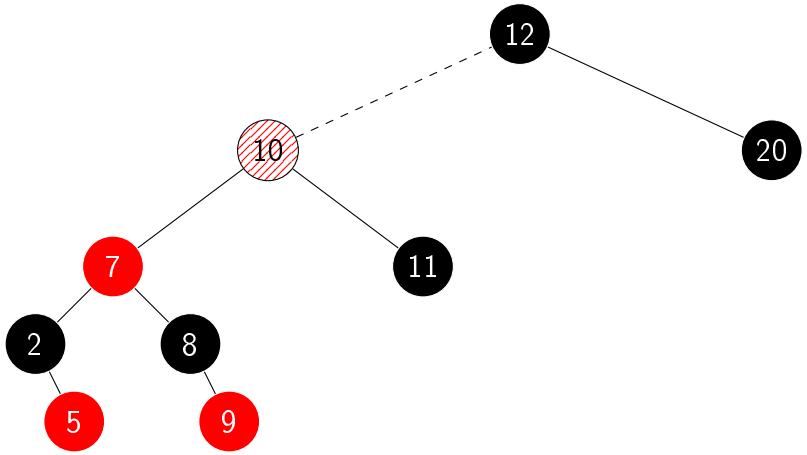
Insertion de 9



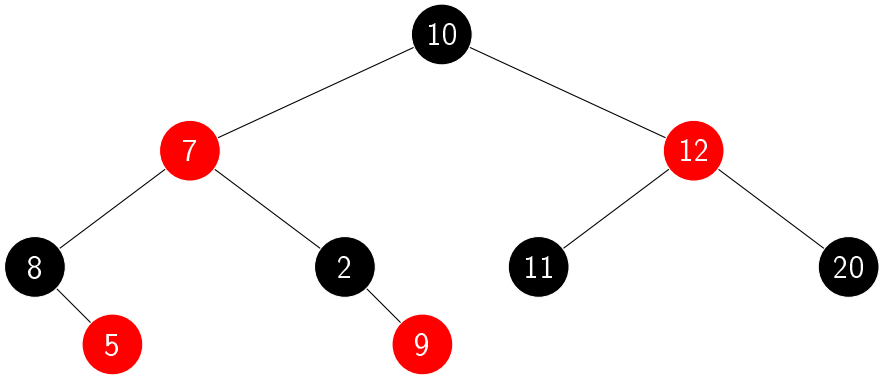
Insertion de 9



Insertion de 9

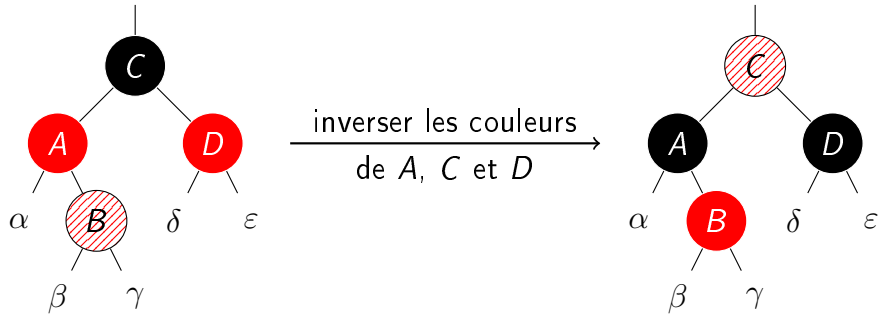


Insertion de 9



Trois cas à gérer : cas 1

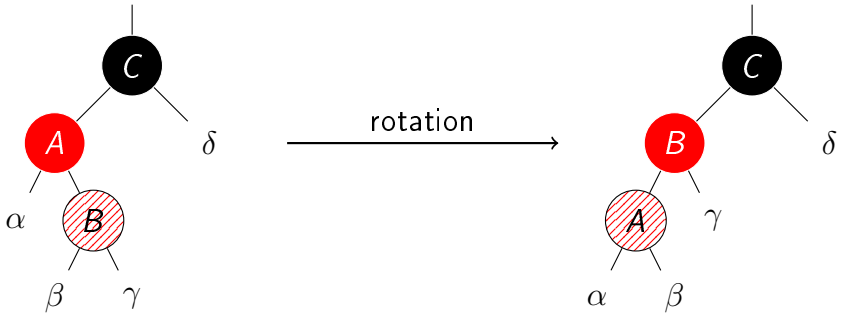
Le père et l'oncle sont tous les deux rouges.
(Dans tous ces cas, les lettres grecques représentent des sous-arbres avec la même hauteur noire.)



Continuer à partir du grand-père si l'arrière grand-père est rouge.

Trois cas à gérer : cas 2

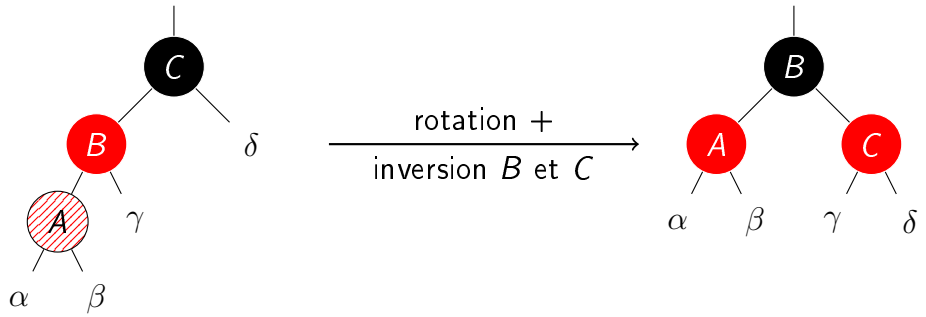
Le père est rouge, l'oncle est noir, et le nœud courant n'est pas dans l'axe père–grand-père.



On effectue la rotation qui convient pour aligner fils, père, et grand-père. Le problème n'est pas résolu, mais on l'a transformé en « cas 3 ».

Trois cas à gérer : cas 3

Le père est rouge, l'oncle est noir, et le nœud courant est dans l'axe père–grand-père.



Après cette correction l'arbre est correct.

Complexité de l'insertion

- On ajoute le nœud avec TreeInsert. $\Theta(h)$
- On applique le cas 1 au plus $h/2$ fois. $\mathrm{O}(h)$
- On applique le cas 2 au plus 1 fois. $\mathrm{O}(1)$
- On applique le cas 3 au plus 1 fois. $\mathrm{O}(1)$

On final, on effectué $\Theta(h) = \Theta(\log n)$ opérations.

RBTreeInsert

```
RBTreeInsert(T, z)
 1  TreeInsert(T, z)
 2  Color(z) ← red
 3  while Color(Parent(z)) = red do
 4     if Parent(z) = LeftChild(Parent(Parent(z))) then
 5        uncle ← RightChild(Parent(Parent(z)))
 6        if Color(uncle) = red then                          ┐
 7           Color(Parent(z)) ← black                         │
 8           Color(uncle) ← black                             │ cas 1
 9           z ← Parent(Parent(z))                            │
10           Color(z) ← red                                   ┘
11        else if z = RightChild(Parent(z)) then              ┐
12           z ← Parent(z)                                    │ cas 2
13           LeftRotate(T, z)                                 ┘
14        else                                                ┐
15           Color(Parent(z)) ← black                         │
16           Color(Parent(Parent(z))) ← red                   │ cas 3
17           RightRotate(T, Parent(Parent(z)))                ┘
18     else comme le « then », en échangeant « Left » et « Right »
```


La suppression

RBTreeDelete() peut aussi être effectué en $\Theta(\log n)$.
Si le nœud à supprimer est rouge, TreeDelete peut être utilisé.
S'il est noir, une procédure de correction de l'arbre doit être appelée ensuite : il y a cette fois 4 cas à distinguer.

Conclusion sur les ARN

Les arbres rouges et noir permettent d'effectuer toutes ces opérations en $\Theta(\log n)$:

- Insert
- Delete
- Minimum
- Maximim
- Successor
- Predecessor

De plus Search se fait en $\mathrm{O}(\log n)$.

L'intérêt sur les tables de hachage est surtout que les éléments sont conservés triés.

Cette structure de donnée est celle utilisée par `std::map` en C++.

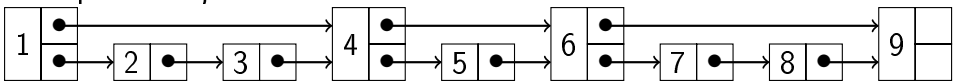
Skip list

Skip list = liste à enjambement.

Généralisation de la structure de liste triée.

Il s'agit d'une structure probabiliste, ayant la même complexité en moyenne que les arbres rouges et noirs (i.e. $\Theta(\log n)$ en moyenne pour toutes les opérations), avec de grandes chances d'atteindre cette moyenne, et plus simple à implémenter.

Exemple de *skip list* à deux niveaux :

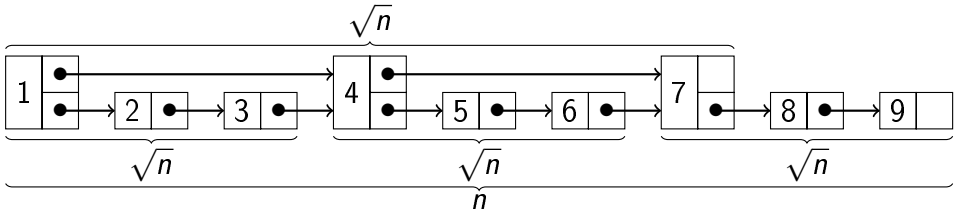


On localise d'abord l'intervale de l'élément recherché dans la liste de plus haut niveau, puis on descend dans la liste inférieure pour affiner la recherche.

Skip list a deux niveaux

Où doit-on connecter les deux niveaux ?

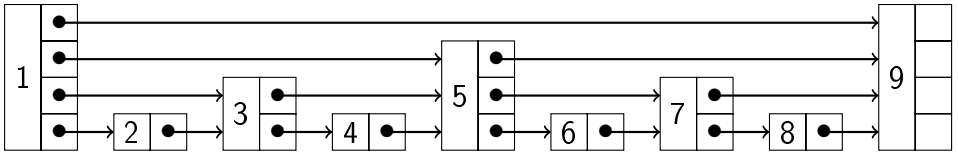
On veut un espacement régulier, mais avec quel espacement ?
Si on note t_1 et t_2 la taille des deux listes. Le temps de recherche d'un élément dans la *skip list* est en $\mathrm{O}(t_1 + t_2/t_1)$. Cette somme est minimale lorsque les termes sont égaux : $t_1 = t_2/t_1$ autrement dit quand $t_1 = \sqrt{t_2}$.



Le coût de recherche est alors de l'ordre de $2\sqrt{n} = \mathrm{O}(\sqrt{n})$.

Skip list à plus de 2 niveaux

- 2 listes : $2 \cdot \sqrt{n}$
- 3 listes : $3 \cdot \sqrt[3]{n}$
- k listes : $k \cdot \sqrt[k]{n}$
- $\log n$ listes : $\log n \cdot \sqrt[\log n]{n} = \log n \cdot e^{\frac{1}{\log n} \ln n} = \log n \cdot e^{\ln 2} = 2 \log n$



L'exemple ci-dessus est une skip list idéale. Les recherches y sont toujours en $\Theta(\log n)$.

Comment réaliser l'insertion ?

Algorithme d'insertion dans une *skip list*

Soit x à insérer dans une *skip list*.

- Effectuer une recherche pour trouver où insérer x dans la liste du bas (niveau 0).
- Insérer x dans la liste du bas.
- Avec une probabilité $1/2$, ajouter x à la liste de niveau 1.
- Si x a été ajouté au niveau 1, avec une probabilité $1/2$, ajouter x à la liste de niveau 2.
- Continuer pour tous les niveaux.

Au final x apparaît

- au niveau 0 avec une probabilité 1
- au niveau 1 avec une probabilité $1/2$
- au niveau 2 avec une probabilité $1/4$
- au niveau k avec une probabilité 2^{-k}

Analyse du coût de la recherche (1/2)

On évalue le côut de la recherche en comptant les déplacement à partir de la fin : à partir du niveau 0, il faut remonter jusqu'au niveau k en se déplaçant vers la gauche quand on ne peut pas monter.

On note $C(k)$ le coût de monter de k niveaux et $p = 1/2$ la probabilité d'avoir un niveau au dessus dans le nœud courant. Deux cas peuvent se produire :

- avec probabilité p on peut monter d'un niveau et il reste à monter $k-1$ niveaux (coût : $1 + C(k-1)$ déplacements)
- avec une probabilité $1-p$ on ne peut pas monter : on se déplace sur la gauche puis il reste à monter k niveaux (coût : $1 + C(k)$ déplacements)

Autrement dit :

$$C(0) = 0$$
$$C(k) = (1-p)(1 + C(k)) + p(1 + C(k-1))$$

Analyse du coût de la recherche (2/2)

$$C(0) = 0$$
$$C(k) = 1/p + C(k-1) = k/p$$

C'est une borne supérieure du coût de monter n niveaux, car si on atteint la tête de liste en allant trop à gauche la probabilité de monter devient alors 1.

Notons $L(n)$ la hauteur d'une *skip list* de n éléments. Le coût pour remonter au dernier niveau est $C(L(n) - 1)$.

Dans notre cas $L(n) = \log n$. On a donc

$$T(n) = \mathrm{O}\left(\frac{(\log n) - 1}{p}\right) = \mathrm{O}(\log n)$$

Complexités des structures de recherche présentées

opération	table de hachage	arbre R. et N.	*skip list*
$p \leftarrow$Search(S, v)	$\Theta(1)$ moy	$\mathrm{O}(\log n)$	$\mathrm{O}(\log n)$ moy.
Insert(S, x)	$\Theta(1)$ am.	$\Theta(\log n)$	$\mathrm{O}(\log n)$ moy.
Delete(S, p)	$\Theta(1)$ am.	$\Theta(\log n)$	$\Theta(1)$
$v \leftarrow$Minimum(S)	$\Theta(n)$	$\Theta(\log n)$	$\Theta(1)$
$v \leftarrow$Maximum(S)	$\Theta(n)$	$\Theta(\log n)$	$\Theta(1)$
$p' \leftarrow$Successor(S, p)	$\Theta(n)$	$\Theta(\log n)$	$\Theta(1)$
$p' \leftarrow$Predecessor(S, p)	$\Theta(n)$	$\Theta(\log n)$	$\mathrm{O}(\log n)$ moy. ou $\Theta(1)$

am. = amorti ; moy. = en moyenne.